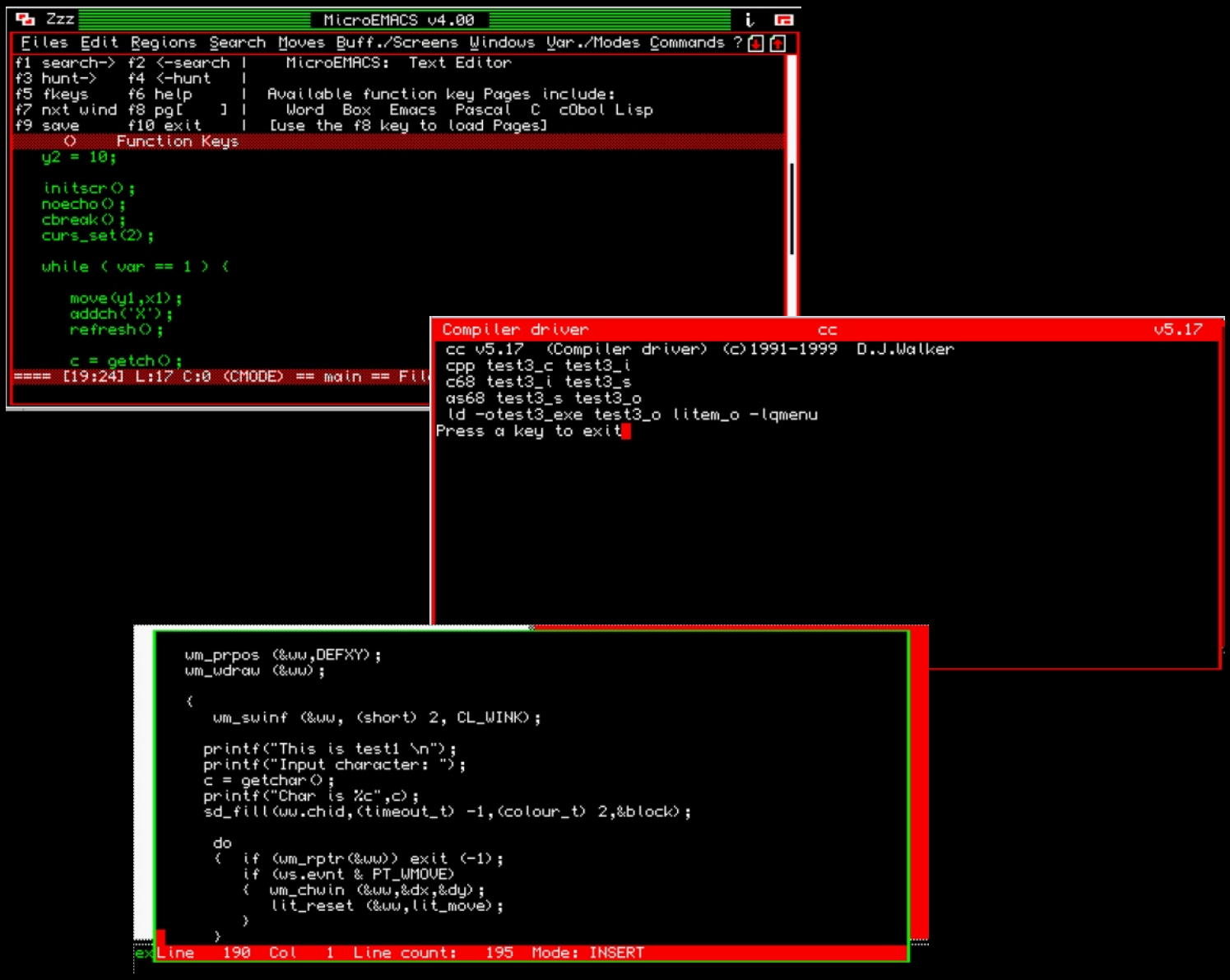
smsqzine

Issue #7 **July 2019**



Special C68 Edition

SMSQzine

table of contents

Published by:
Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

SMSQzine is published as a service to the Sinclair QL community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open Source Tools:

- OpenOffice
- Scribus
- Gimp
- SMSQmulator

Editorial

I have been using C68 for a number of years, but this is the first time I've tried using it for Pointer Environment programs. Once I started tinkering with C68, this lead me on a path looking at a number of C68 topics. The articles are mostly in an order that I took, first with QL graphics with C68, then looking into the QPTR example programs, and then doing some of my own Pointer Environment programs, etc.

There is a lot that comes with C68. There are quite a number of libraries available to the C68 programmer to use. There a few front ends for C68 that I have only looked at. I'm find that I'm back to QED for my editor of choice, but MicroEmacs has some advantages if doing some serious C68 work. It has a C highlight mode that will highlight the C syntax. It is partially mouse driven. It has a whole macro language is you need to do some serious work on C code.

Considering that I'm not all that ANSI-fied my self, I've not explored how ANSI C68 is and what it takes to port code from the rest of the world the C68. I do know that it is more of an ANSI compiler than a K&R compiler.

Hopefully the articles are interesting and maybe that will encourage some of you to try working with C68.

Curses and the QL

Curses is a Unix programming library that is designed to make writing screen-based programs easy on text-based interfaces. It is designed to take the terminal information, such as the terminal type, its size, etc, and allow for standard commands to use the terminal. Programs that use curses the 'top' command and games like Rouge.

Curses was create by Ken Arnold and was written for BSD Unix. A version was created by Mark Horton at AT&T. In 1993, ncurses (new curses) was released. Sometime in the 1990's, curses was ported to the QL. In reviewing the source code, I don't see any mention of who did the port. From the source code, the AT&T version was used in the port.

Curses on the QL is used with the C68 C compiler and the library comes with the distribution disk. The C68 source code disks also have the curses source code.

Curses lets you do screen commands, like CLS and AT. Most Unix/Linux programs just treat the screen as an infinite roll of paper and old text just scrolls up the page as new text appears at the bottom. Very much like using an old ASR-33 teletype machine. Curses changes the output to be more like a true screen, where text can be placed at any location.

Here is a sample bit of curses code:

```
     initscr();
     addstr("Enter a Character (then
hit enter) ");
     refresh();
     getchar(x);
     move(10,10);
     addchr(x);
     refresh();
```

This will prompt a user to enter a character, move the cursor to a postion of 10,10, and then print the character to the screen. The refresh() command updates the screen. Sort of like doing a PAUSE on the ZX81 when in FAST mode. The screen is updated in the background and then the refresh() call displays it.

The test program I wrote for curses was to move the X character from left to right on the screen, making sure to wipe out the old character before printing the new one. The core part of the program is:

```
   initscr();

   for (i = 1; i <= 10; i++)
      for (x = 10; x <= 40; x++)
         move(15,x);
         addch('X');
         refresh();
```

```
          for (a = 1; a <=
1000000; a++)

          move(15,x);
          addch(' ');
          refresh();
```

The initscr() function creates a new default window. All of the commands following use the default window. There are ways to open multiple windows and address each command to a specific window.

One thing to note about the move command, it takes the Y argument first, where as in most functions that use the screen, use the X argument first. It just takes a bit of getting use to.

The long loop in the middle of the program is a delay. Without it the program would run too fast for you to see the movement.

Here is another example that shows some more curses commands:

```
wnd = initscr();
getmaxyx(wnd,r,c);
addstr("  Size of Window ");
printw("Width is : %d ",c);
printw("Height is : %d ",r);

move(15,10);
addch('X');
attrset(A_UNDERLINE);
addch('U');
attrset(A_BLINK);
addch('X');
attrset(A_BOLD);
addch('B');
attrset(A_REVERSE);
addch('R');
refresh();
```

Note that how the initscr() function is used to assign a value to a variable. The getmaxyx needs to know the pointer to the window, so I had to get the pointer. The addstr() prints a string, but only a string. If there is a need to print a variable, then the printw() command is needed. It is very similar to the standard printf() command.

The last bit of the program is to test the different text attributes. I found that underline worked as did reverse. Blink did nothing and bold had the same output as reverse.

Here is another program based on the first one:

```
 initscr();
 noecho();
 cbreak();
 curs_set(0);

while ( var == 1 )

    move(y1,x1);
    addch('X');
    refresh();

    c = getch();
    if ( c == 'j') y2 = y1+1;
    if ( c == 'k') y2 = y1-1;
    if ( c == 'l') x2 = x1+1;
    if ( c == 'h') x2 = x1-1;
    if ( c == 'a') var = 0;

    for (a = 1; a <= 500000; a++)


    move(y1,x1);
    addch(' ');
    refresh();
```

```
        x1 = x2;
        y1 = y2;
```

This program looks for input from the user (either the H, J, K, or L keys) and moves the X on the screen. For those familiar with Unix, these are the standard keys for moving left, down, up and right. The X is printed to the screen and as keys are hit, the X is moved around the screen. To exit, the A key is hit.

The cbreak() command tell curses to not wait for a return character before getting the input. It functions more like INKEY$ than INPUT. The noecho() command tell curses to not echo the key hit to the screen. The curs_set() command is supposed to make the cursor invisible, but it does not appear to be working.

Finding good documentation on curses is not easy. I have a copy of "Programming with Curses" by John Strang, a book from O'Reilly & Associates. It is a good introduction to curses, but there is a whole lot of other commands that are left out. If you want a good introduction this book does a pretty good job.

If found the best documentation to be the documents on ncurses. There is probably a difference between ncurses and the version ported to the QL, but ncurses is close enough.

The document can be found at:
https://invisible-island.net/ncurses/man/ncurses.3x.html

Look for the section "Routine Name Index" and there you will find links to the individual man pages for the commands.

To know what curses commands are available on the QL, I have gone through the sources and created a list of each command. This list is included in the .zip file that goes with this issue. There are close to 270 commands in the QL version. Most of these can be found in the ncurses documentation. Some commands may not be exactly the same spelling as the ncurses version, but the names are close enough.

The curses.h file in the C68 distribution is helpful to look at, as it shows a number of the attributes that are used with curses. This is how I found A_BOLD and A_REVERSE. It also does list information about the commands, but in a slightly less readable format.

When using a curses program, you must set up the QL environment for it. You have to have the Environment Variables extension loaded (ENV_BIN). Then there are two ENV commands to set up. The location of the terminal information has to be defined:

```
SETENV "TERMINFO="win1_C68_LIB"
```

And what the terminal will be:

```
SETENV "TERM=qdos"
```

So the terminfo_qdos file will be loaded.

If it is not clear, the need for having curses on the QL is the ability to port curses-based program from Unix/Linux to the QL. There are a few older Unix games that are curses-based that could be ported to the QL. The game Rogue has been ported by Jerome Grimbert, later updated by Thierry Godefroy. Elvis, the clone of the vi test editor, ported by Dave Walker, also uses curses.

I have used curses to write a ASCII implementation of my cellular automata program. I am finding that there is a '0' that appears on the screen with no associated command to put it there, so there could be some bugs in the curses library.

C68 with QL Graphics

I have been porting my Cellular Automata program to as many lanuages as I can on the QL (plus ZX81, T/S 2068 and Spectrum). When using other C compilers on the QL (Small-C, DP C & QC) the QL graphics

commands are treated just like any other C function calls, using the same file pointer. When I started writing the program for C68, it was clear that C68 was different and it was not obvious how to mix standard C input and output functions with QL graphics.

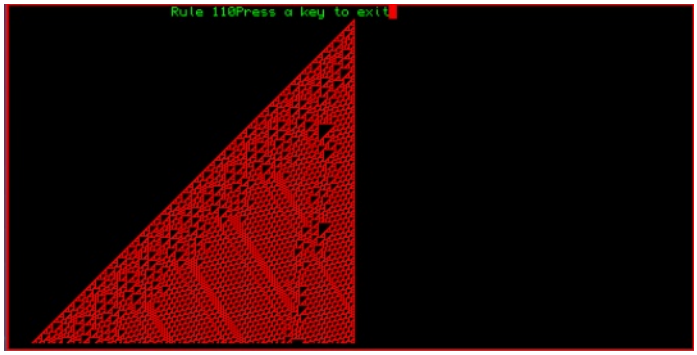
There is some discussion in the C68 documentation about the three different I/O levels in C68 and how the QL native layer is not the same as the C layer. I had posted a query on the Sinclair QL Forum and got some answers and then I sort of sat on the issue for a

while. I looked at the code for CBzone to see how that program did it and got a few ideas.

I was trying to open my own window and then using that window for both levels of I/O (C and QL). I thought I had something working but it only sort of looked like it was working. I was trying to use the initial console, thinking that an window open call for "CON_" was referring to the default console, when in fact I was opening a new console. Luckily, Tobias on the QL Forum mentioned that the device for the default console as 'stdout'. Once I knew that I could then convert the C file pointer for 'stdout' to a QL channel id.

When writing to the screen with a C function, since it is the default console, I don't need to refer to a file pointer. The command printf is used instead of fprintf, which needs a file pointer. When I wanted to use QL graphics, then I used the channel id that I got from 'stdout'.

In my code, I use _condetails to set up the default console window to be the size that I want. I then use fgetchid(sdtout) to get the channel id of the console window. My test program will be included in the .zip file for this issue.



Once I had this working, I was able to get the Cellular Automata program running with C68, which means I could check off another language / compiler on my list.

C68 Documentation

I like documents, and I like it when they look good. Printing a document from Quill to a dot-matrix printer does not cut it for me. With a number of QL documents, I have converted them to Open Office and made them look better. I have done this with the C68 documents. I had done this previously, but there was some documents that I had not added.

The end result is two documents, one on C68 and one on the C68 libraries. I've included 10 of the libraries available for C68. The libraries:

LibANSI, LibC68, LibQDOS, LibSMS, LibQPTR, LibM (math library), LibUnix, LibCurses, LibVT (terminal emulation) and LibDebug.

Since I like hard copy, I will print both of these documents out, but I will make them available as PDF files. I did some C68 programming while traveling. I did not bring the hardcopy along, but had the PDF files on my netbook. Using SMSQmulator, I could switch between programming and checking the

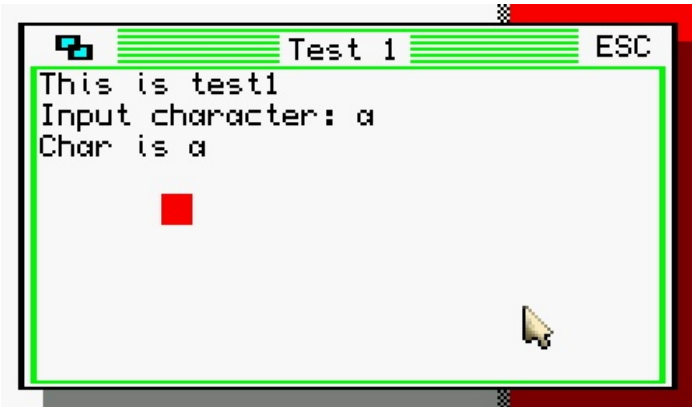
C68 documentation. It worked rather well.

The documentation is available at http://swensont.epizy.com/.

Programming the Pointer Environment with C68

I've been thinking about programming with the Pointer Environment for almost 20 years. I've decided to take a small dip in the pool and see how hard it would be to do it using C68.

The QPTR distribution for C68 comes with a number of example programs, that start with a simple window and progress to more advanced programs, adding features as the examples move along. The examples come with documentation written by Tony Tebby and go over the important details of each of the examples. I found the document hard to follow without looking at what each example looked like. To solve this issue, I compiled all of the examples to see what they look like and to added them to the tutorial document. This document will be made available.



When I had started looking into the Pointer Environment years ago, I started working on a document that was an introductory text on the individual parts of the Pointer Environment (Loose Item, Sub-Window, etc) to try and understand what they are and how they work together. I never finished that document and never really got a grasp of the different parts of the Pointer Environment.

As I was looking through the examples, I thought it would be faster to use the examples as a template for a pointer environment program. Find one of the examples that comes closest to what you need, and then modify it to fit your needs. The size of the window can easily be changed in the code. If there is a Loose Item that you don't need, then remove it.

Now, this might not work of you are working on a fairly complex program, but if you are just working on something fairly simple, then starting with one of the examples can save a lot of time trying to start from scratch.

As a test I copied the wind_2c_c example and copied it to test1_c. I then moved the eg_h file from the distribution disk and put it in win1_c68_INCLUDE to go with all the other _h files. I also put all _o files into win1_c68_LIB directory.

I edited test1_c to reflect the new name. I also changed the Window Title to be what I wanted. The wind_2c_c example displays a number of lines to the window and the waits for either a MOVE or ESC action. I altered the program to just print out two sample lines of text, just to show that it behaves different than the example.

I compiled it with:

```
   exec cc;"-o win2_test1_exe
win2_test1_c win1_c68_LIB_litem_o"
```

With environment variables it is possible to set the working directory for the files.

This program worked and I did little more than edit a few lines in the PE code to reflect the size or color of the window. I did a second example where I used what I had learned about QL graphics with C68 and added a block command to put a block on the screen, along with some text.

XTC68

XTC68 is a version of C68 ported to other computers, such as Linux or Windows. The early version was

also ported to DOS and NT, but the most recent version had not touched those OS's.

The advantage of XTC68 is the ability to do all of the development work and compiling on a different operation system, one with possibly better coding tools that QDOS or SMSQ/E and to compile the program on a faster platform.

Back in the 90's when most people has BBQL's, doing the coding and compiling on another computer was much faster. The BBQL was a slower system, most used floppies which had limited space, etc. Putting everything on a MS-DOS, Windows or Linux computer with a faster processor, larger hard drive, etc. made for faster development.

The current version of XTC68 has recently been touched, so it is more or less actively maintained. This version is hosted on GitHub and to get a copy you will need to have Git installed on your computer. I'm using Linux, so I will be providing instructions on how to get it up and running on Linux.

The first thing is to get the source:

```
% git clone
https://github.com/stronnag/xtc68
```

This will download the source and put it into a directory called xtc68.

```
% cd xtc68
% make
```

Once in the source code directory, run the make command. One thing to note, XTC68 is a 32-bit application and the compiler will compile it as a 32-bit application. If you are running on a 64-bit version of Linux, you will need to install the 32-bit libraries.

XTC68 needs the first disk of the C68 distribution to have the header and include files. This is 424frun1.zip (found on Dilwyn's website). Download that zip file, put it in the support directory and extract all of the files.

The last step is to run the installation script (from within the main xtc68 directory):

```
% ./install.sh
```

Now you are ready to compile programs for the QL. I found a simple "Hello world" program online, dumped it to the xtc68 directory, and them made sure it would work under C68. I executed XTC68 just like C68, except that compiler is called "qcc" instead of "cc". This also differentiates it from the "cc" GCC compiler on my Linux system.

```
% qcc -o hello.exe hello.c
```

Note that there is no need to use the underscore as the extension seperator but the standard dot will do.

Within a few seconds the compiler finshed and there was hello.exe on disk.

The next step is getting the binary to the QL and putting the dataspace into the header part of the executable. This is done with a version of Info-Zip modified for the QL, but compiled for Linux. The version on Dilwyn's site will not run and I had to get an undated version from Graham. This will be included on the zip file for this issue.

```
% qlzip -Q2 hello.zip hello.exe
hello.c
```

This will create the hello.zip file the source and executable files. It will also add the dataspace to the executable file.

On the QL, I used the normal unzip to extract the files. The QL version of unzip will convert the . to a _, so that hello.exe will become hello_exe. Once on the QL, I was able to execute hello_exe and it ran just fine. As a test I took the same source code and compiled it with C68 and basically got the same executable.

With the advent of faster systems and faster

emulators, the speed issue may no longer be a factor. The issue now might be one of programming tool familiarization and what one prefers. When I write for the QL, I use the QL with QED as my editor. Granted it is not mouse driven and I'm used to a mouse driven editor, I just adjust. Someone else might find that they cannot live without the real Emacs and want to develop using that editor.

There are a number of IDE's for Linux that should be able to be modified to use XTC68. Just change the call to 'cc' to 'qcc'.

A New Make

When I was looking at the C68 QPTR example programs, I was going to use Make to compile them. The examples did come with a Makefile to make it easy to compile the whole set. When I ran make, it brought up a console window and then nothing. It took me a few seconds to realize that SMSQmulator emulation had locked up. I could use the SMSQmulator toolbar options, but the emulation was locked up. I reset the emulator and tried again. I got the same results.

I knew that make had worked in the past, so I found another .win file and found an older version of make. This version, 2.0d, ran fine. The version that came with C68 4.24f, 2.0f, was not fine.

I contacted Wolfgang to let him know about the issue and see if he could shed any light on make. He found that it also crashed QPCII. He said the program "writes to memory starting at address 0, so it overwrites the 'ROM' - no wonder nothing will work anymore." He found that two subroutines did this.

Since C68 comes with a source distribution, I decided to see if I could recompile make and see if that worked. I extracted the source zip file and put it on my local disk. I thought I would try to compile it with XTC68. The compile went fine, I zipped it with qlzip, but once I got it to SMSQmulator, it would not run. I tried setting the data space with a tool, but also did not work.

I created a new .win file, dumped the make source code on it and tried to compile it by hand. That did not go so well. I'm not sure I put the source files in the right order.

Since I had a working make, let's see if make can make make. I copied the version 2.0d binary over to my C68 disk and ran make on the make Makefile. There were a few warnings, but within a minute I had a new make 2.0f binary. When I ran it, it ran fine. EXEC make;"-h" gave the help screen.

I'm not sure what caused the original version of 2.0f to be bad. I even want back and pulled a fresh copy from the C68 distribution zip files and it still crashed SMSQmulator.

This new version of make will be included in the zip file for this issue.

Xmenu with C68

Xmenu was written by Jerome Grimbert. It is a form of Qmenu for C68, written entirely in C. The main functions are popup windows for making a selection, enter a string, or selecting a file. Xmenu is comprised of two header files and a binary library file, libxmenu_a. To use Xmenu, the two include files are copied to the normal C68 include directory. The libxmenu_a is copied to the normal C68 library directory. When using C68, the library is used by adding "-lxmenu" at the end of the call to cc.

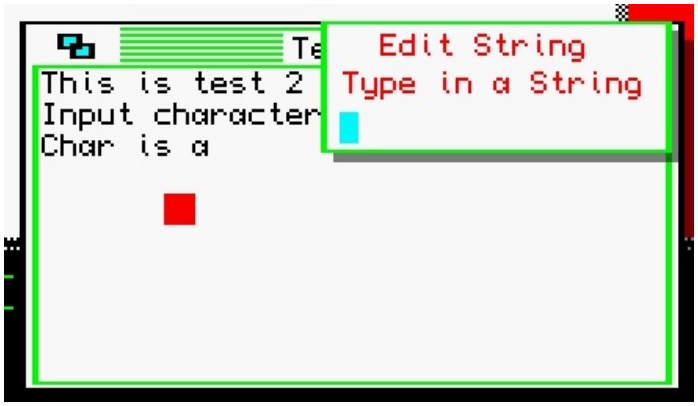
The documentation for Xmenu is limited and is only the two include files. The function calls are detailed, but there is no example code showing what sort of values the calls can have.

The functions are:

```
Item_Select()
List_Select()
Radio_Select()
Message_Report()
String_Edit()
```

```
XDialog()
Menu_Button_Text()
Menu_Button_Logo()
......
```

There are comments about each of the functions listed in xmenu_h that details the arguments that it needs. Each function needs a structure that contains a number of colors for the popup window. The



structure, ColourSet, is defined in xmenudef_h and has 14 items. Without any documentation I guessed that each of the items is a color value and made some guesses at what values are needed.

I did a quick test using the previous test1_c from tinkering with the Pointer Environment. I added the two include files and then added a structure based on colourSet. Since colourSet is created with a typdef, there is no need to start the definition like this:

```
structure colourSet mycolor =
```

But just use colourSet as it aleady been defined a structure:

```
colourSet mycolor
```

For my test, I wanted to get some string input. I used String_Edit.

```
x =
String_Edit(&mycolor,"Title","Enter
String","",15,*str);
```

The second string is the title of the popup. The second string is the explanation for what is being inputted. The third string is the default value. For my needs, I left it as blank. The next arguement is the length of the input string. The final value is the variable for the string.

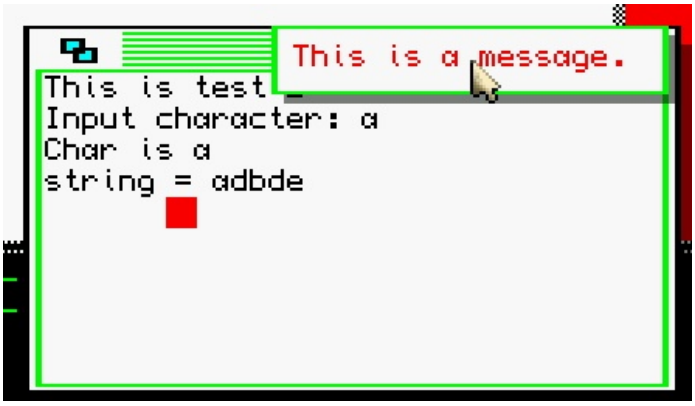
After compiling, the program ran fine and I got the two popups that I was expecting.

The next thing I tried was Item_Select where one item is selected from a list of items. The code is like this:

```
x =
Item_Select(&mycolor,"Title","Select
Item",3,"One","Two","Three");
```

The popup will create a list of three items that are mouse selectable. The return value is which one the user selected. In the above case "One" would return 1 as it is the first item in the list.

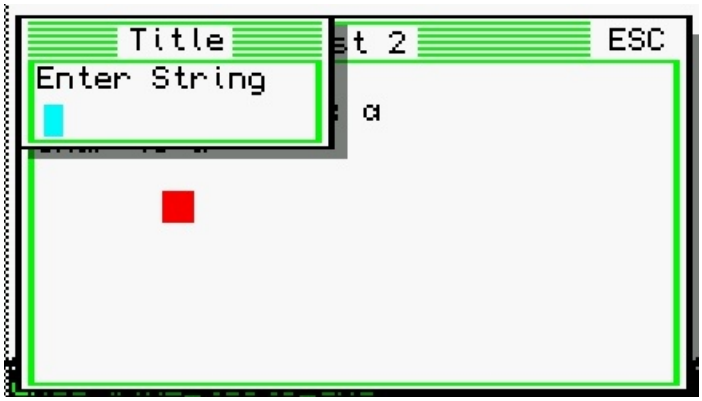
After compiling, I ran through the program, but when Item_Select was supposed to come up, the top half of the screen went black and SMSQmulator emulation just halted.



I next tried the Radio_Select call and got the same behavior. To see if it was an issue with SMSQ/E or SMSQmulator, I tested the program out on uQLx and it also crashed. Maybe it was the compiler. I compiled the program using XTC68. It also crashed SMSQmulator. It was dawning on me that there is a bug in Xmenu. There is a demo binary that is a separate download. I downloaded it and tried it. It worked fine and did not crash the emulator. I can only guess that it was compiled with a slightly older

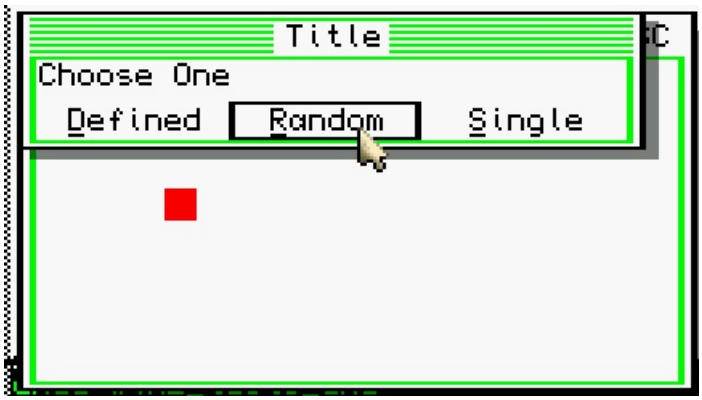
version of C68.

For now, I'll forgo using Xmenu until I can dig into this some more.



C68 with Qmenu

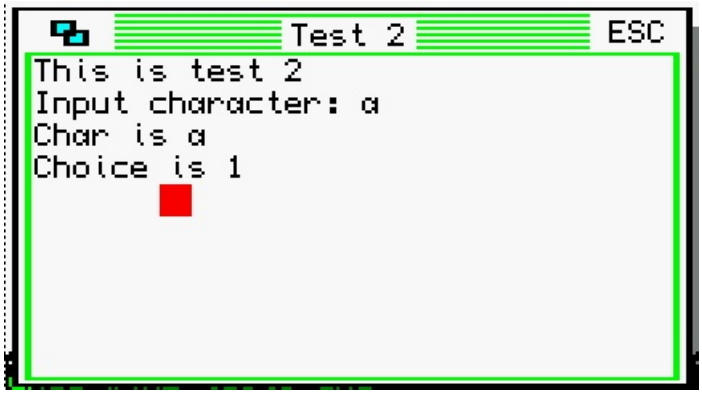
Another way of adding pointer environment popups to C68 programs is the library that is used to call Qmenu from C. Qmenu is pointer environment menu system (popups) written by Jochen Merz for both assembly and for SuperBasic. It is loaded during boot and any program can call it. The C68 library allows for C programs to call Qmenu.



The C68 Qmenu library was written by Christopher Cave and Johnathan Hudson, with some fixes by Thierry Godefroy. It comes with the library in binary format, two header files, a bunch of examples and a document file.

Adding Qmenu calls to my program was fairly simple. There is a line that increased the stack for the progarm that is needed. The menu_h header file as added to the program.

When looking at the calls, the documentation talked about three of the arguments that are used on almost any call. Looking at the examples, make it clearer exactly they were talking about.



My first test was with ReadString, which provides a popup for entering or editing a string:

```
   x = ReadString("Title","Enter
String","",10,str,0,0,-1)
```

The third empty string is just the default string. After the str string variable, the 0,0 is an x and y coordinate to locate the popup in the original window. Since I said 0,0, the popup will be in the upper left corner. The last value is the color scheme to use. The value -1 means to use the default for Qmenu.

When I compiled this, it worked the first time.